Controlling knowledge transfers in distributed algorithms.Application to deadlock detection

Jean-Michel Hélary, Aomar Maddi, Michel Raynal

IRIA

CENTRE DE RENNES

IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 493

CONTROLLING KNOWLEDGE TRANSFERS IN DISTRIBUTED ALGORITHMS

APPLICATION TO DEADLOCK DETECTION

Jean-Michel HELARY
Aomar MADDI
Michel RAYNAL

Mars 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

Publication Interne n° 278
32 pages
Janvier 1986

CONTROLLING KNOWLEDGE TRANSFERS IN DISTRIBUTED ALGORITHMS

APPLICATION TO DEADLOCK DETECTION

Jean-Michel HELARY, Aomar MADDI, Michel RAYNAL
IRISA Avenue du Général Leclerc 35042 RENNES FRANCE

ABSTRACT : Distributed algorithms are fundamentally characterized by the lack of a global knowledge which could be instantaneously cast by a process. Thus, to realise a distributed computation, it is necessary to provide processes with some knowledge acquisition protocols. Several distributed algorithms have been devised in such terms of knowledge transfers. In order to reduce the number of exchange messages required by knowledge transfers, we introduce the concept of control knowledge and present protocols based on it which reduce this number.
The concept is then used to devise distributed deadlock-detection algorithms. When the wait-for graph is complete, the resulting algorithm requires only 2(n-1) messages to allow one of the processes to know whether it is or not definitively waiting.

RESUME : Une des caractéristiques fondamentales des algorithmes distribués est l'absence d'une connaissance globale que pourrait capter instantanément un des processus qui réalisent l'algorithme. En conséquence, l'élaboration d'un calcul distribué nécessite généralement de mettre à la disposition de chacun des processus des protocoles d'acquisition d'informations (ou connaissances) qu'ont les autres processus. Plusieurs algorithmes distribués ont ainsi été conçus en termes de transferts de connaissances entre processus. Afin de réduire le nombre de messages échangés nous introduisons le concept de connaissance de contrôle et présentons des protocoles fondés sur son utilisation qui réduisent le nombre de transferts.
En guise d'illustration le principe proposé est appliqué à la détection de l'interblocage dans un système distribué. Dans le cas où le graphe des attentes est complet, l'algorithme obtenu ne requiert que 2(n-1) messages pour permettre à l'un des n processus de savoir s'il est ou non définitivement bloqué.
Ce rapport généralise les résultats présentés dans le rapport INRIA n°427. Outre le cas où le graphe des blocages est statique, on montre que l'algorithme de détection proposé résiste à l'évolution dynamique de ce graphe des blocages ; un concept de période d'observation est introduit à cet effet.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

CONTROLLING KNOWLEDGE TRANSFERS IN DISTRIBUTED ALGORITHMS

APPLICATION TO DEADLOCK DETECTION

JEAN-MICHEL HELARY, AOMAR MADDI and MICHEL RAYNAL
IRISA Avenue du Général Leclerc 35042 RENNES FRANCE

Key words

Distributed algorithms, deadlock, knowledge, control knowledge.

1. Controlling knowledge acquisition

A distributed algorithm is made of a finite set of communicating sequential processes exchanging messages (RAYNAL, 1985a). Such a system is fundamentally characterized by the lack of a global state which could be instantaneously cast by a process (LAMPORT, 1978). Furthermore, a given process initially has access but to local information (or

knowledge) and the implementation of a distributed computation involves protocols to gain knowledge from the other processes.

1.1 Knowledge acquisition

Several algorithms have been designed which let a process gain some knowledge to which it initially has no access. Thus we can find in (BOUGE, 1985) a distributed algorithm allowing each process, initially knowing only the set of its neighbours (the processes which it can exchange messages with), to obtain a global knowledge of the the communication network; in (SCHNEIDER, 1985) a distributed algorithm which, under the same initial assumptions, allows every process to get all the others initial knowledge; in (CHANDY and LAMPORT, 1985) an algorithm which provides each process with a record from every other's previous state, the union of which constitutes a global coherent previous system state.

As can be seen from these examples, most distributed problems can be cast in terms of initial or gained knowledge transfers between processes. Algorithms devoted to these problems use the same basic technique to implement information transfers and to obtain the desired knowledge : they are known as wave algorithms (SCHNEIDER, 1985). The wave principle is very simple : a process, wishing to get some knowledge, broadcasts an appropriate message towards each of its successors (processes to which it can send a message). Each of them receives that message and rebroadcasts it along with its own local information, and so on. Messages are thus sent through the processes, carrying the knowledge stored in by the latter when they were visited, and the knowledge brought in by a message is increased by successive accumulations of local information. If the directed communication network is strongly connected, the process which initiated the request message will finally receive the desired knowledge from its predecessors (processes from which it can receive a message).

Termination is the main problem arising in this technique. In the case when cycles not passing through the initiating process exist in the communication network, messages may run forever along such cycles. This

phenomenon can be avoided if we observe that a single tour around the cycle gives the sum of all the information stored in by the processes belonging to the cycle. Consequently, when a process receives a request message, having received before other messsages related to the same request, it checks whether the knowledge brought in by this message is the same as the one recorded previously (which was obtained by summing up its local knowledge to those of the previous messages), in which case none of the processes belonging to the cycle around which the message has just circulated increased it : broadcasting to the successors is thus unnecessary.

1.2 Controlling knowledge transfers

Algorithms based upon this technique involve, for every process, the transmission of the knowledge obtained from its predecessors, increased by its local knowledge, towards its successors. A number of messages of various lengths (depending upon the gain of transmitted knowledge) have to be transmitted in order to allow an initiating process to obtain a desired knowledge.

We propose a technique which reduces this number of exchanged messages, by systematically taking into account the fact that every process does know its successors. Let us draw the attention to the situation previous to any communication : each process only knows local information, namely initial knowledge and set of successors.

A process may then broadcast towards its successors a knowledge request message containing two kinds of features : one is related to the knowledge which the initiator process wishes to obtain (data-knowledge), the other to the progress of this knowledge through the processes (control knowledge). Before rebroadcasting a message to its interlocutors, a process increases on the one hand the knowledge required by the initiator, on the other hand the control knowledge, adjoining the set of its interlocutors to the latter. Thus a process knows, upon receiving such a message, a subset of processes which have received or are about to receive (depending upon the speed of the messages) the information related to the same request issued from a

given process. According to the kind of desired knowledge, a process can eventually decide not to rebroadcast messages towards those of its interlocutors belonging to this subset.

This principle will be applied below to the classical control problem of deadlock detection for whole or part of a system. The second paragraph presents the problem and the third one the assumptions to be made on the behaviour and topological features of the communication; algorithms based upon the proposed principle are derived in the fourth and sixth ones; they allow a given process to determine whether or not it belongs to a deadlock subset at some time, according to hypotheses done about the static/dynamic behaviour of the wait-for graph.

2. Deadlock subset

A system of communicating processes can be designed by a non-directed graph whose vertices are processes and edges are direct communication lines existing between pairs of processes. We are interested in the problem of deadlock detection for whole or part of processes in the system, due to mutual impossibility to communicate. This is a typical distributed control problem : it can be designed by a directed graph $G=(X, E)$ whose vertices are processes and an edge from P_i to P_j means that P_i is waiting for an event (e.g sending or receiving a message of an underlying computation) which can be produced only by P_j.

G is called *Wait-for graph*. If processes are endowed with non deterministic control structures (HOARE, 1978 ; ICHBIAH et al, 1983), a process P_i can eventually wait for an event among several which can be produced respectively by $P_j,\ldots,P_k$, in which case the edges $(P_i,P_j),\ldots,(P_i,P_k)$ belong to the wait-for graph.

The situation in which these processes are definitively blocked (deadlock) is made quite clear by introducing the notion of *deadlock subset*. Let's recall that a process is definitively blocked if and only if, on the one hand it is waiting for an event (possibly among several ones) and on the other hand all the processes from which this event is expected (in a non deterministic way) are also definitively blocked and there is no message in transit between these processes.

This situation is resumed in the following definition :

B is a deadlock subset in the wait-for graph G=(X,E) if and only if :

(i) $B \subseteq X$

(ii) $\Gamma(B) \subseteq B$ (where Γ is the function : set of successors)

(iii) $\forall P_i \in B : \Gamma(P_i) \neq \emptyset$

(iv) there is no message in transit.

As can be seen from this definition, deadlock detection is not equivalent to the detection of a cycle in the wait-for graph (because of non-determinism in waiting for an event among several possible ones).

The problem of deadlock detection including non deterministic waiting has been stated and solved by a wave algorithm in (CHANDY et al.,1983). In a distributed context, it can be formulated as follows : a process, whose inactivity is due to the expectation of events which can be produced by other processes, wonders : "am I definitively blocked ?"; in other words, does this process belong to a deadlock subset? It is easy to see that a process P_i belongs to such a set if and only if $\Gamma^*(P_i)$ has no exit vertices (vertices without successors) and there is no message in transit in the subgraph spanned by $\Gamma^*(P_i)$ (by Γ^* we mean the reflexo-transitive closure of Γ, that is to say P_i together with the set of its descendants)

In fact, if P_i belongs to a deadlock subset B, one has $\Gamma(P_i) \subseteq B$ whence $\Gamma^*(P_i) \subseteq B$ and thus, according to the definition of a deadlock subset, the following assertion holds :

$$\forall P \in \Gamma^*(P_i) : \Gamma(P) \neq \emptyset$$

Conversely the following assertion

$\forall P \in \Gamma^*(P_i)$: $\Gamma(P) \subseteq \Gamma^*(P_i)$ is always true and furthermore, if $\Gamma^*(P_i)$ has no exit vertices, $\Gamma^*(P_i)$ satisfies the definition of a deadlock subset.

Thus, the problem of deadlock detection for a given process is equivalent, from the point of view of this process in a distributed context, to the acquisition of a knowledge which it doesn't initially own : does it belong or not to a deadlock subset.

The rest of this paper is devoted to algorithms which enable a process

to get an answer. Depending upon hypotheses on G, two cases will be examined : in fourth paragraph, we suppose that G is a static graph; in other words, we suppose that condition iv) is implicitely verified. Although this can appear as a non realistic assumption, this case allows us to illustrate our principle in a simple manner, being free from time constraints; moreover, it shows how this principle can be used in many problems having a static underlying graph (calculus of global topological properties). In paragraph five,this algorithm is proved, whilst in paragraph six we show how to fit the algorithm in the realistic case of a dynamic wait-for graph : condition iv) has to be explicitely verified in order to avoid any false deadlock detection.

3. Hypotheses

The hypotheses are related to the message communication in a distributed context : sense of communications (connexity hypotheses) and properties related to message transfers (behaviour hypotheses).

The requesting process has to broadcast its request through the wait-for graph. The requested knowledge can reach it back if some connexity assumptions hold; but such assumptions cannot be reasonably made, for the wait-for graph is arbitrary. We will therefore suppose that the edges of the wait-for graph, although directed, are able to convey messages both ways. This bi-directional assumption is to be found in all the knowledge acquisition algorithms in the case when no particular hypothesis can be made on the topology of relations between processes (see the diffusing computation in (DIJKSTRA and SCHOLTEN, 1980; MISRA and CHANDY, 1982; CHANDY and MISRA, 1982a and 1982b)).

Any message sent by one process to another is supposed to be received correctly after an arbitrary but finite delay (messages are neither lost nor altered).

In the static case, no assumption is made on the sequencing of messages between two communicating processes, whilst in the dynamic case we will suppose that messages sent by a process are received in the same order. Finaly, messages received by a process from different ones at the same time are arbitrarely ordered.

4. The detection algorithm in the static case

In this case, the wait-for graph cannot change, or, equivalently, condition iv) for a deadlock subset is supposed to be implicitely verified.

The algorithm to be established lets a process P to know whether or not it belongs to a set B verifying i), ii), iii); in other words, we solve a problem related to a gobal topological property of the graph, namely : "does every path issued from P lead to a knot?", following the terminology of (MISRA and CHANDY, 1982).

4.1. Principle : a control tree

A process wondering whether or not it belongs to a deadlock subset broacasts a knowledge request, through a request message sent to its successors in the wait-for graph; the latter processes increase this message and rebroacast it towards their respective successors. Together with the progress of this wave, a control tree is built up, allowing the answer to reach back the initiating process.

This tree is built according to the following rules : upon receiving the first request message, a process records the sender's identity, which becomes its father in the tree, then broacasts the request to its successors which in turn will become its sons; a process without successors immediately sends back the answer *no* to its father (it is not blocked); upon the following occurences of the request, the answer *yes* is immediately sent back to the sender; as soon as a process gathers all the answers expected from its sons, they are synthetized and the resulting answer is sent back to the father. When the root (the initiating process) has received all the expected answers, the requested knowledge is obtained.

This principle of control tree was introduced by DIJKSTRA and SCHOLTEN in (DIJKSTRA and SCHOLTEN, 1980); it has been since widely used in a number of distributed algorithms. If we apply the principle stated in 1.2 (controlling knowledge transfers) to the build up of the control tree, we can derive distributed algorithms involving less messages, whereas keeping a bounded length.

In (CHANDY et al, 1983) such a control tree is used to allow a process to determine whether or not it is definitively blocked. If the wait-for graph is complete (any process is waiting for an event from any other one), the number of messages is equal to $n(n-1)$ requests, and as many answers, that is a number in $O(n^2)$. In the same configuration, our proposed algorithm involves only $2(n-1)$ messages.

4.2. Messages sent

There are two kinds of messages sent between processes in this algorithm : *request* messages carry on the request of knowledge by a given process together with the control knowledge ; *answer* messages bring the state of partial knowledge back up the tree. The first ones run in the sense of the wait-for graph's edges, the second in the opposite way.

A *request* message has the following structure : *(request,z,i)* where the control knowledge z is equal to the set of processes which have been or are about to be reached in by the request message, thereby controlling the transfers.

i denotes the sender of the message.

An *answer* message is of the form : *(answer,b)*

where b is equal to one of the two values *yes* or *no* (corresponding respectively to the boolean values *true*, *false* to which we can apply boolean operators to obtain the resulting knowledge).

4.3. Local context of a process

Local variables used by each P_i in order to maintain computational and communication contexts are defined as follows :

- $received_i$ is boolean, initially equal to *false*; it is switched to *true* as soon as a request is received.
- $resp_i$ takes values in { *yes*, *no* }; initialized to *yes* it does record the partial knowledge, related to the request.
- $succ_i$ denotes the set of processes to which P_i will send the request; initially, it is equal to the set of the successors of P_i in the wait-for graph : it is a local knowledge. This variable is used to

implement the controlling transfer principle : it is updated at the time of the first occurence of a message *(request,z,j)* according to *z*.

- $pred_i$ is the identity of the first process from which P_i has received a request message; it is the *j* component in this message. It occurs in the dynamic build up of the control tree and will consequently be used to carry back the knowledge brought in by the answers.
- $nbrep_i$ is a non negative integer variable, counting the number of expected answers.

NOTE. The algorithm can be fit either to the case where several processes wonder whether they belong or not to a deadlock subset or when the request is repeated several times by the same process unwaiting the corresponding answers. All what has to be done is to identify uniquely the requests and to change up local variables into arrays : each entry is then related to a given initiator process. Unique identification of requests is obtained if we consider the pair :
(initiator identity,request number) as the request identifyer. The algorithm will be thereby modified. (request numbers issued from a given process define an increasing sequence. Similar methods are used for instance in (CHANDY et al.,1983) for deadlock detection and in (SCHNEIDER et al.,1984) for messages broadcasting).

4.4. The algorithm

Upon the first occurence of a request message, P_i performs the following action : if it has no successors, it sends the answer *no* back to the sender; otherwise it checks up whether the set of its successors is included in *z*, in which case it sends the answer *yes* back to the sender, on the contrary it records its father's identity (namely the sender) then broadcasts towards those of its successors not belonging to *z* the request message modified by updating *z*; thereafter it does wait for answers from these processes and as soon as it gets them (or as soon as it is able to conclude, see the note after the algorithm) it sends back the appropriate answer to its father. By receiving subsequent request messages, P_i will immediately send the answer *yes* back to the

emettor.

The text of P_i is given below in an "event driven" form as in (RAYNAL,1985b) : actions to be undertaken upon the arrival of a given message are described.

P_{init} will denote the process initiating the knowledge request.

upon receiving a message *(request,z,j)*

```
    do
1     if received then send (answer,yes) to P_j
2       else
3         received := true;
4         if succ = ∅
5           then (* resp := no *) send (answer,no) to P_j
6           else
7             if succ ⊆ z
8               then send (answer,yes) to P_j (* resp:= yes*)
9               else
10                pred := j;
11                succ := succ - z;
12                nbresp := card(succ);
13                ∀k ∈ succ: send (request,z ∪ succ,i) to P_k
                fi
            fi
        fi
    od
```

upon receiving a message *(answer,b)*

```
      do
14        resp := resp and b;
15        nbresp := nbresp - 1;
16        if i ≠ init and nbresp = 0
17           then send (answer,resp) to P_pred
18        fi
19    od
```

P_{init} issues the detection by simulating the reception of the message *(request,{init},init)*.

The termination occurs (i.e. there are no more messages related to the P_{init}'s request in transit) as soon as P_{init} receives the answers from all its successors. Whether or not P_{init} belongs to a deadlock subset is then characterized by the value *yes* or *no* taken by the variable $resp_{init}$.

NOTE. This knowledge can eventually be obtained by P_{init} before the temination occurs. In fact, if P_{init} receives a message *(answer,no)* from one of its successors, it can conclude that it does not belong to a deadlock subset because the variable $resp_{init}$ is switched to *no* (line 14) and by the properties of the boolean operator *and*, will keep this value up to the end of the computation: the assertion *resp=no* is stable. By the way, this remark holds for any process expecting answers; thus we can improve the behaviour of the algorithm by the following description of the action to undertake by receiving an *answer* message :

upon receiving a message *(answer,b)*

```
do
  if resp = yes
     then resp := b;
          nbresp := nbresp - 1;
          if i ≠ init and (nbresp = 0 or resp = no)
              then send (answer,resp) to P_pred
          fi
          (*else the answer has already been sent, when
            resp has been switched to no; discard the message*)
  fi
od
```

Clearly, through this modification the response time of the algorithm (time lying between request sending and knowledge completion by P_{init}) is improved, but the number of messages exchanged remains the same. However, all the answer messages received by processes whose variable *resp* has been switched to *no* will be discarded.

5. Proof

The proof consists in two parts :

- Termination : when the initiator process (afterwards denoted by P_0) has issued a knowledge request, it gets an answer after a finite time.
- Partial correctness : P_0 belongs to a deadlock subset if and only if the answer is *yes* .

In order to complete this proof, we introduce the concept of performing tree, to which some properties are related.

5.1 Performing tree

While the algorithm performs, a control tree is built up

dynamically : it is specified by the variables $pred_i$; according to the speed of messages, this tree can be different from an execution to an other. Likewise, we consider a performing tree A, built as follows :

(i) The root is labelled P_0

(ii) As soon as a process P_i receives a message *(request,z,j)* a node labelled P_i is added to the tree; its father is a node labelled P_j (the message 's sender).

Moreover, in this tree a node labelled P_i is a leave if and only if one the three assertions holds :

i) $received_i$=*true* upon the reception of the message (line 1) : there is already a node labelled P_i in A

ii) $received_i$=*false* and $succ_i \subseteq z$ upon the reception of the message (line 7) (first reception of a *request* message by P_i)

iii) $received_i$=*false* and $succ_i = \emptyset$ upon the reception of the message (line 4).

In the first two cases, P_i sends back immediately the answer *yes* to its father : we shall say that P_i is a non-determining leave. In the third case, it sends back the answer *no* and we shall say that P_i is a determining leave.

Let's emphasize that the control tree actually built up by the variables $pred_i$ (line 10) is a subtree, with the same root, of the performing tree (the latter's leaves do not exist in the control tree).

5.2 Properties of the performing tree

5.2.1 Proposition 1

When P_i receives a message *(request,z,j)* the control knowledge *z* contains :

a) vertices labelled $P_0,\ldots,P_j,P_i$ situated along a path $u = [P_0 \ldots P_j P_i]$ in G

b) vertices belonging to $\bigcup_{P \in [P_0 \ldots P_j]} \Gamma(P)$

Proof

We proceed by recurrence on the length of the path. The result is trivial for P_0 (length 0). Suppose that it's true for P_j, the sender of *(request,z,j)* to P_i.

Then, according to lines 11 and 13, we have :

i) $P_i \in \Gamma(P_j) - z_j$ (where z_j is the control knowledge received by P_j), otherwise P_i wouldn't have received that message.

ii) $z = (\Gamma(P_j) - z_j) \cup z_j$ (control knowledge sent by P_j

The control knowledge z received by P_i thus contains :

- z_j , that is, by the recurrence hypothesis :
  * vertices $P_0,\ldots,P_l,P_j$ belonging to a path $[\, P_0 \ldots P_l P_j \,]$
  * vertices belonging to $\bigcup_{P \in [P_0 \ldots P_l]} \Gamma(P)$
- The vertex P_i since $P_i \in \Gamma(P_j) - z_j$
- $\Gamma(P_j)$ since $\Gamma(P_j) = (\Gamma(P_j) - z_j) \cup (\Gamma(P_j) \cap z_j) \subset z$. QED.

5.2.2 Proposition 2

A node labelled P_i appears at least once in the performing tree A if and only if $P_i \in \Gamma^*(P_0)$ (Γ^* is the reflexo-transitive closure of Γ).

Proof

Let's assume that $P_i \in A$. If i=0, then trivially $P_0 \in \Gamma^*(P_0)$; else, let P_j be the father of P_i in A; by construction, $P_i \in \Gamma(P_j)$; thus, stepwise upon the branch passing through P_i in A we obtain $P_i \in \Gamma^*(P_0)$.

Conversely, let's show by recurrence on the integer k that the following assertion holds :

$$P_i \in \Gamma^k(P_0) \Rightarrow P_i \in A.$$

This is trivial for k=0 ($P_0 \in A$). Suppose that it holds up to k-1 and let $P_i \in \Gamma^k(P_0)$; then $\exists\ P_j \in \Gamma^{k-1}(P_0)$ such that $P_i \in \Gamma(P_j)$. By the recurrence hypothesis, $P_j \in A$. Consider the first node labelled P_j, added to A when the first message *(request,z,1)* reached P_j. We have :

$\Gamma(P_j) = (\Gamma(P_j) - z) \cup (\Gamma(P_j) \cap z)$, so that if $P_i \in \Gamma(P_j) \cap z$ then P_i already belongs to A (by proposition 1) and if $P_i \in \Gamma(P_j) - z$, it will be added to A as a son of P_j (lines 10,11,13) since messages are not lost.

This completes the recurrence, and the conclusion follows from the definition

$$\Gamma^*(P_0) = \bigcup_{k=0}^{\infty} \Gamma^k(P_0).$$ QED.

5.2.3. **Corollary 2.1**

$\forall\ P_j \in A$, we have $\Gamma(P_j) \in A$. This result was obtained during the proof of proposition 2.

5.2.4. **Proposition 3**

Every process $P_i \in \Gamma^*(P_0)$ appears at most once as label of an inner node (i.e. node which is not a leave) in A.

Proof

A node labelled P_i will generate a son only if, by receiving a request message, the assertion *received*$_i$*=false* holds for the process P_i; but this boolean variable is switched to *true* when the first occurence of a request message reaches P_i (line 3) and keeps on this value afterwards. QED.

5.2.5. **Corollary 3.1**

When a process $P_i \in \Gamma^*(p_0)$ appears several times as label in A, the labels of its respective fathers are mutually different.

Proof

By proposition 3 a process can generate a son in A at most one time. According to line 13, it will send thereafter at most one request to each of its successors. QED.

5.3. **Termination**

Theorem 1 The algorithm will terminate in finite time

Proof

Observe first that the program of each process (whose number is

finite) has no loop. Thus, an infinite computation time would result only from the exchange of message. But, by proposition 3 and corollary 3.1, the performing tree is finite and, by construction, every node receives exactly one request message, after a finite time since the transmission delays are finite. Each node will send back an answer to its father after a delay δ. This delay is finite. In fact, this is trivial in the case when the node is a leave; for an inner node, a non finite δ value would mean that this node indefinitely waits for an answer from one of its sons, whence there would exist a branch issued from this node upon which every process waits for ever. But it can't happen since each branch has a finite length and a leave answers after a finite delay. From this follows that P_0 gets the answers from its sons in a finite time. QED.

5.4 Partial correctness

5.4.1 Proposition 4

At the termination of the algorithm, for each process $P_i \in \Gamma^*(P_0)$ the two following assertions are equivalent :

i) $resp_i = no$

ii) A node labelled P_i exists on a branch with determining leave.

(Let's recall that a leave is said to be determining if the process which labels it has no successors in the wait-for graph (line 4)).

Moreover, if one of these two assertions hold, then P_i does not belong to a deadlock subset.

Proof

i) $\Rightarrow$ ii) if $resp_i = no$, at least a branch passing through a node labelled P_i ends down with a leave which has sent back the answer *no*; this leave is thus a determining one. (The assertion $resp_i = no$ is stable : if it is *true* at a point of the computation, it remains true afterwards).

ii) $\Rightarrow$ i) If a node labelled P_i belongs to a branch with a determining leave, it can be :

- either the leave itself, in which case $resp_i$ is switched to *no*

(line 5)

- or an inner node, in which case the determining leave has sent an answer *no* which goes back on the branch up to the node labelled P_i, where the following computation is performed : $resp_i$=*no* and ...=*no*.

This shows the equivalence of the two assertions. Moreover, if one assertion holds, P_i has a descendant without successors (namely the determining leave considered above), therefore it does not belong to a deadlock subset. QED.

5.4.2. **Proposition 5** (pseudo-reciprocal of 4)

If $P_i \in \Gamma^*(P_0)$ does not belong to a deadlock subset, then the performing tree has at least a branch with a determining leave (P_i possibly belongs to this branch).

Proof

If the conclusion is false : all the branches of tree have non-determining leaves. It follows that the set B of processes belonging to the tree verifies :

$\forall P \in B : \Gamma(P) \neq \emptyset$ (non-determining leaves)

$\forall P \in B : \Gamma(P) \subseteq B$ (corollary 2.1)

Thus B is a deadlock subset; moreover, by proposition 2, P_i belongs to the tree since $P_i \in \Gamma^*(P_0)$; the contradiction with hypothesis follows.QED.

5.4.3. **Theorem 2** (partial correctness)

Upon termination of the algorithm, the following assertion holds for the process P_0, initiator of the knowledge request :

$resp_0$ = *no* $\Leftrightarrow$ P_0 doesn't belong to a deadlock subset.

Proof

If $resp_0$=*no* then P_0 doesn't belong to a deadlock subset, by proposition 4.

Conversely, if P_0 doesn't belong to a deadlock subset, the performing tree has at least a determining leave, by proposition 5. This leave sends back the answer *no*, which is propagated on the branch up to the

root P_0, thereby making the variable $resp_0$ be switched to the value *no*. By stability, the assertion $resp_0$ = *no* still holds at termination. QED.

6. Detection algorithm in the dynamic case

6.1. The situation

Let's examine now the same problem as before, in which we release the assumption that condition iv) is *a priori* fulfilled. During the deadlock computation issued by a process, messages belonging to the underlying computation may be in transit; the reception of such a message by a process P_i will modify the wait-for graph G : all the edges leaving P_i are then removed from G. According to this new fact, the response received by a process initiating a deadlock detection must fulfill the two following conditions :

C1 : if P_{init} receives the answer *no*, the conclusion is : P_{init} did not belong to a deadlock subset at the time t_0 when P_{init} issued the detection. (Nothing can be said about the situation of P_{init} upon the reception of the answer : being not definitively blocked is not a stable property!).

C2 : if P_{init} receives the answer *yes* the conclusion is : P_{init} does belong to a deadlock subset at the time t_f when this answer is received, and consequently at any time $t \geq t_f$ (this property is stable) (CHANDY and LAMPORT, 1985; CHANDY and MISRA, 1985).

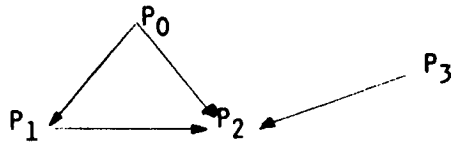
Thus, we point out the fundamental feature : be careful to false deadlock detection!

The algorithm established in the static case must be fitted to make sure that C1 and C2 are verified.

The case of C1 makes no problem : in fact, if P_{init} belongs to a deadlock susbet at the time t_0, then the subgraph of G spanned by $\Gamma^*(P_{init})$ is static since processes belonging to $\Gamma^*(P_{init})$ cannot issue messages, and there are no messages in transit. The situation is thus similar to the one examined before, and proofs remain valid.

Let's examine now the case of C2, through the following example :

At time t_0, P_0 issues a deadlock detection, G being as follows :



Consider a sequence of events, belonging either to the underlying computation or to the deadlock detection issued by P_0 driven by the algorithm of paragraph 4, occuring in chronological order (related to an abstract global clock):

t_0 : P_0 sends *(request,z,0)* to P_1 and P_2, $z=\{P_0,P_1,P_2\}$.

t_1 : P_1 receives *(request,z,0)*, thus sends *(answer,yes)* to P_0 according to lines 7-8 (for $P_2 \in z$).

t_2 : P_2 sends a message *m* to P_1.

t_3 : P_2 waits for a message from P_3 :the edge (P_2,P_3) is created in G.

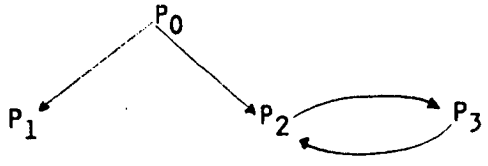
t_4 : P_2 receives *(request,z,0)*, thus sends *(request,z2,2)* to P_3, $z2 = z \cup \{P_3\}$.

t_5 : P_2 receives *(answer,yes)* from P_3, thus sends *(answer,yes)* to P_0.

t_6 : P_1 receives *m* from P_2 : the edge (P_1,P_2) is removed from G.

t_f : P_0 received answers *yes* from P_1 and P_2.

Thus P_0 declares itself definitively blocked, whereas the wait-for graph is at time t_f :



showing that P_0 does not belong, at that time, - and consequently at any time $t \leq t_f$ - to a deadlock subset. This example shows that a deadlock detection driven by our preceding algorithm (derived in a static context) must be adapted to overcome false deadlock detection.

6.2. Modification of the algorithm

The reason for which the previous algorithm has to be modified is the following : according to the control of knowledge transfers, a process P distinguishes between two kinds of successors when reached in

by the first request message : the controlled successors (that is belonging to the control knowledge *z* brought by this message), and the others; the first ones are not considered as successors by P, and the difficulty arises if some messages of the underlying computation issued by such successors are still in transit when P is able to send back an answer to its father and this answer is equal to *yes*.

This failure can be overcome by using one of the following remedies :

* Removing the control of knowledge transfers : this increases the number of messages. (The result is an algorithm similar to the one devised in (CHANDY et al., 1983)).

* Making a new assumption on the maximum transfer delay of messages, namely : message delays are bounded up by Δ (whether they belong to underlying or control computation).

We shall now derive a modified algorithm based upon this second approach.

We will also make the following assumption : messages sent on a line are received in the order they were sent (no overtaking).

From now on, the word "message" will mean a message belonging to the underlying computation.

There are three modifications in the text of P_i :

* The first one consists to introduce a delay 2Δ, between the first reception of a request and the time at which P_i will be allowed to send back an answer to its father; this delay will be applied in the case where P_i has at least one controlled successor and the answer to be sent is equal to *yes*.

* The second one is : if P_i receives a message while it is waiting for sending back an answer to its father then P_i sends immediately the answer *no* , switches $resp_i$ to *no* and subsequently discards all expected answers, if any.

* The third one is : if P_i receives a *(request,z,j)* message and $received_i = true$ (line 1) then it sends back immediately *(answer,*$resp_i$*)* (instead of *(answer,yes)* as previously) to P_j.

Remark 1 .

An obvious consequence of the second modification can be stated as follows : if a process P_i answers *yes* back to its father $pred_i$ then, at that time, $\Gamma(P_i) \neq \emptyset$; in other words, no message m can be subsequently sent by P_i unless it receives a message m' from one of its successors. This remark will be used in the proof derived below.

6.3. **Proof**

According to the considerations of 6.1, only the condition C2 has to be proved. In what follows, we will denote by :

t_0 : the time at which P_0 issues the detection,

t_i : the first time at which P_i receives a request *(request,z,j)*,

t'_i : the time at which P_i sends back an answer to its father $pred_i$ (replying to the first request).

Proposition 6

For any process $P_i \in \Gamma^*(P_0)$, having a controlled successor P_k ($P_k \in z$) at time t_i the following holds :

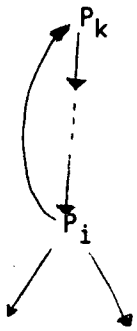
* either no message issued by P_k will reach P_i after time $t_i + 2\Delta$,

* or P_k sends back the answer *no* to its father.

Proof

We consider the two possible cases for such a P_k.

1) P_k is an ascendant of P_i in the control tree (and thus $P_k \in z$). At time t_k, P_k has sent a request along the branch to which P_i belongs, which means that P_k is blocked at time t_k (it has successors in G). Moreover $t_k < t_i$.



Suppose P_k sends a message *m* to P_i at time *t*; *m* will reach P_i at time $t' \leq t + \Delta$.

If $t < t_k$, we have thus :

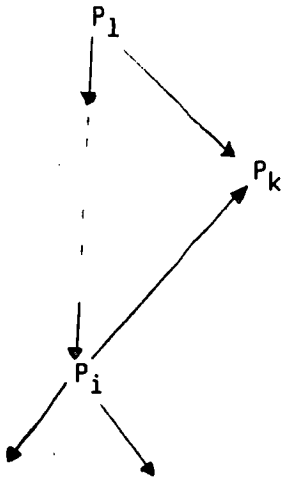
$t' \leq t + \Delta < t_k + \Delta$ and thus *m* will reach P_i before time $t_i + 2\Delta$.

If $t > t_k$, P_k must have been deblocked by the reception of a message expected from one of its successors in G, in which case the answer *no* is sent back by P_k

2) P_k is successor of an ascendant P_l of P_i in the control tree (and thus $P_k \in z$).

At time t_l, P_l sends a request to P_k and along the branch to which P_i belongs, and we have :

$$t_l < t_k \leq t_l + \Delta, \qquad t_l < t_i$$



Suppose P_k sends a message *m* to P_i at time *t*; *m* will reach P_i at time $t' \leq t + \Delta$

If $t < t_k$ we have : $t' < t_k + \Delta \leq t_l + 2\Delta$ in which case *m* reaches P_i before time $t_i + 2\Delta$,

If $t \geq t_k$, the situation is similar to the case 1): P_k sent back the answer *no*. QED.

Proposition 7

If P_i receives an answer *yes* from a non controlled successor P_i (one of its sons in the performing tree), the following holds :

* either 1) no message will be sent by P_k to P_i after P_k has sent the answer *yes* to P_i,

* or 2) $i \neq pred_k$ and P_k answers *no* to its father $pred_k$, or P_k answers *yes* to its father and subsequently receives a message from one of its successors.

Proof

By hypothesis, P_i sends a request to P_k , and, at time t , P_k sends back the answer *yes* to P_i.

We show that, if 1) is false, then 2) holds. Let t' be the time at which P_k sends a message m to P_i. Since messages cannot overtake, we have $t' > t$. Two cases have to be considered :

i) $pred_k \neq i$ (in other words : the request received by P_k from P_i is not the first one).

If P_k sends an answer *yes* to $pred_k$, at time t'_k, then :

* either $t'_k < t$, in which case P_k is blocked at time t (otherwise P_k should have answered *no* to P_i) : a message m' must have reached P_k in the interval $]t, t']$ since P_k is not blocked at time t' , and therefore m' has been received by P_k after time t'_k;

* *or* $t'_k > t$, in which case, from the remark 1 at the end of 6.2 , the second part of 2) is verified.

If P_k sends an answer *no* to $pred_k$, then by hypothesis, $pred_k \neq i$ and 2) is trivially verified.

ii) $pred_k = i$: it follows from remark 1 that a message has been received by P_k after time when P_k sent back *yes* to its father P_i. QED.

Theorem 3

If P_{init} receives the answer *yes* then it belongs to a deadlock subset at the time t_f when it receives this answer.

Proof

$\Gamma^*(P_{init})$ is a deadlock subset : axioms i), ii),iii) of the definition are derived as in the static case (theorem 2); it remains to show that there are no message in transit at time t_f. This is obvious from proposition 6 and 7, and from the first modification : let P_i be any process belonging to the performing tree. By hypothesis, P_i sends back *yes* to its father, at a time $t'_i \geq t_i + 2\Delta$; hence no message issued by a controlled successor of P_i is in transit at time t'_i (proposition 6); moreover, by induction, if a message issued by a non controlled successor of P_i was in transit at time t'_i, we could construct a sequence of processes $P_i,P_j,\ldots,P_k,P_l,\ldots$ such that P_k receives a message from P_l after sending *yes* to its father (proposition 7); it follows from the first part of this demonstration that P_l must be a non controlled successor of P_k whence P_l differs from the preceding terms in the sequence; now this sequence is finite; the contradiction follows. QED.

7. Conclusion

A general knowledge acquisition technique, based upon transfers control, has been stated; the number of messages sent between the processes realizing together a distributed algorithm can be reduced. We emphasize that the length of messages remains bounded.

The basic principle of this technique is very simple : never do again what has been done, unless necessary. (Such a principle can be found in sequential algorithmic : dynamic programming,...). Here, it is applied to message transfers.

This technique can be applied to problems concerned with knowledge acquisition by a process, provided that the operators involved in the knowledge computation be associative, commutative (the order in which partial knowledge is accumulated and transmitted is unrelevant to the complete expected knowledge) and idempotent (adjunction of a partial knowledge to an other one in which it is contained doesn't modify the latter).

A deadlock detection algorithm has been proposed which illustrates the above technique. The length of messages is bounded up by the number n of processes, and their number remains less- or equal in the worst cases- than in the other detection algorithms. Moreover, due to the boolean character of desired knowledge, the answer is obtained as soon as possible (in other words, it can be obtained even before the detection termination). This last property holds every time that, besides the above algebraic properties of operators, the type of knowledge has an absorbing element (here the value *no*).

Deadlock detection field is interesting on several accounts with regards to distributed algorithmic. Actually, a quiescent property has to be cast by a given process : "am I definitively blocked ? " : the question is essentially relative to the initiating process. In that sense, it differs from the termination problem, for which the question, althoug issued by a given process, involves all of them : "are we all terminated ?".

In these two problems, difficulty results from the dynamic nature of relations between processes : to deal with it, a notion of observation period related to each process has to be introduced; two hypotheses relative to message transmission have been made in order to implement this notion : bounded delays and no overtaking. Results obtained agree with those in (CHANDY and MISRA, 1985) as for observation periods and with those in (WUU and BERNSTEIN, 1985) : in this last paper a delay of 2Δ is shown to be a sufficient condition to avoid false deadlock detection in a distributed context.

REFERENCES

BOUGE L.,1985
Symmetry and Genericity for CSP Distributed Systems.
Rapport de Recherche, LITP, Universite de PARIS 7, (Mai 1985), 22p.

CHANDY K.M., LAMPORT L., 1985
Distributed Snapshots : determining global states of distributed systems.
ACM TOCS,Vol. 3,1, (February 1985), pp. 63-75.

CHANDY K.M., MISRA J., 1982a
Distributed computation on graphs : Shortest paths Algorithms.
Comm. ACM, Vol. 25,11, (November 1982), pp. 833-837.

CHANDY K.M., MISRA J., 1982b
Termination Detection of Diffusing Computations in CSP.
ACM TOPLAS, Vol. 4,1, (january 1982), 37-43.

CHANDY K.M., MISRA J., HAAS L., 1983
Distributed deadlock detection.
ACM TOCS, VOL. 1,2, (may 1983), pp. 144-156.

CHANDY K.M, MISRA J., 1985
A paradigm for detecting quiescent properties in distributed systems.
Springer Verlag, NATO ASI Series, VOL F13, 1985 , pp. 325-341.

DIJKSTRA E.W., SCHOLTEN C.S., 1980
Termination Detection for Diffusing Computations.
Inf. proc. Letters, Vol. 11,1, (August 1980), pp. 1-4.

HOARE C.A.R., 1978
Communicating Sequential processes.
Comm. ACM, Vol. 21,8, (August 1978), pp. 666-677.

ICHBIAH J., et al., 1983
Reference Manual for the ADA programming language.
ANSI/MIL-STD, 1815A, (January 1983).

LAMPORT L., 1978
Time, Clocks and the Ordering of Events in a distributed System.
Comm. ACM, Vol. 21,7, (July 1978), pp. 558-565.

MISRA J., CHANDY K.M, 1982
A distributed graph algorithm : knot detection.
ACM, TOPLAS, VOL. 4,4, (october 1982), pp. 678-686.

RAYNAL M., 1985a
Towards a better understanding of Distributed algorithms.
proc. of an int. workshop on Distributed computing, Newcastle, (April 1985).

RAYNAL M., 1985b
Algorithmes distribués et protocoles.
Eyrolles, (octobre 1985), 160p.

SCHNEIDER F.B., GRIES D., SCHLICHTING R.D., 1984
Fault-tolerant broadcast.
Science of Computer, VOL. 4,1 (1984), pp. 1-15.

SCHNEIDER F.B., 1985
Paradigms for distributed programs.
in distributed systems, Springer-Verlage,LNCS 190, (1985)
pp. 431-480.

WUU G.T, BERNSTEIN A.J, 1985
False deadlock detection in distributed systems.
IEEE, Trans. on Soft. Eng., VOL. SE11,8, (August 1985), pp. 820-821.

PI 272 **Architecture pour les opérations geométriques en synthèse d'images par facettes**
François Charot, Franck Rousée – 24 pages ; Novembre 85.

PI 273 **Madmacs : a new VLSI layout macro editor**
Patrice Frison, Eric Gautrin – 12 pages ; Novembre 85.

PI 274 **Détection de pannes et reconfiguration automatique**
Michèle Basseville – 26 pages ; Novembre 85.

PI 275 **Estimation de l'ordre d'un processus Arma à l'aide de résultats de perturbations de matrices**
Jean – Jacques Fuchs – 40 pages ; Décembre 85.

PI 276 **Detection and diagnosis of changes in the eigenstructure of nonstationary multivariable systems**
Michèle Basseville, Albert Benveniste, Georges Moustakides, Anne Rougée – 44 pages ; Décembre 85.

PI 277 **Optimum Robust Detection of Changes in the Ar Part of a Multivariable Process**
Michèle Basseville, Anne Rougée, Georges Moustakides, Albert Benveniste – 48 pages ; Décembre 85.

PI 278 **Controlling knowledge transfers in distributed algorithms. Application to deadlock detection**
Jean – Michel Hélary, Aomar Maddi, Michel Raynal – 32 pages ; Janvier 1986.

Jean – Michel HÉLARY

Aomar MADDI

Michel RAYNAL

CONTROLLING KNOWLEDGE TRANSFERS IN DISTRIBUTED ALGORITHMS APPLICATION TO DEADLOCK DETECTION

Publication interne
n° 278

Janvier 1986